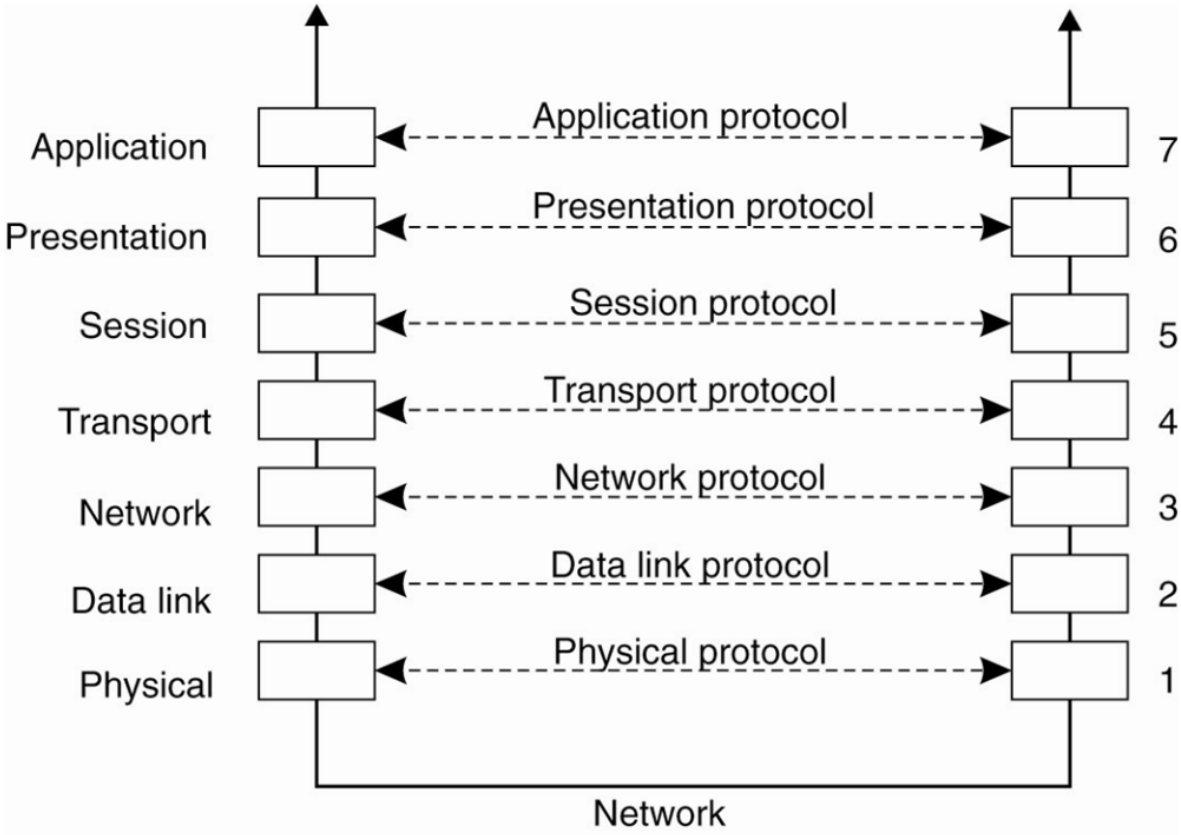
Communication

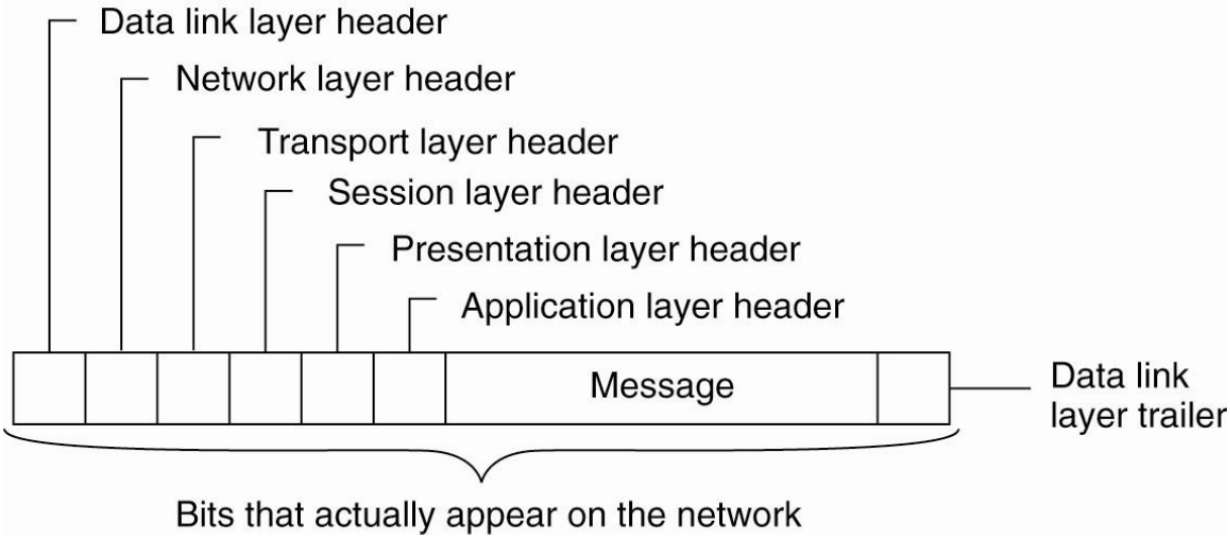
Fundamentals

The OSI model

Almost all communication in distributed systems is based on the **OSI model**:



Every message that is sent in a network based on the OSI model is encapsulated according to the protocol layers:



Middleware as a protocol layer

Middleware includes common services and protocols that can be used by many different applications:

- Marshalling (and unmarshalling) of data (necessary for integrated systems)
- Naming protocols, to allow easy sharing of resources
- Security protocols
- Scaling mechanisms (replication, caching...)

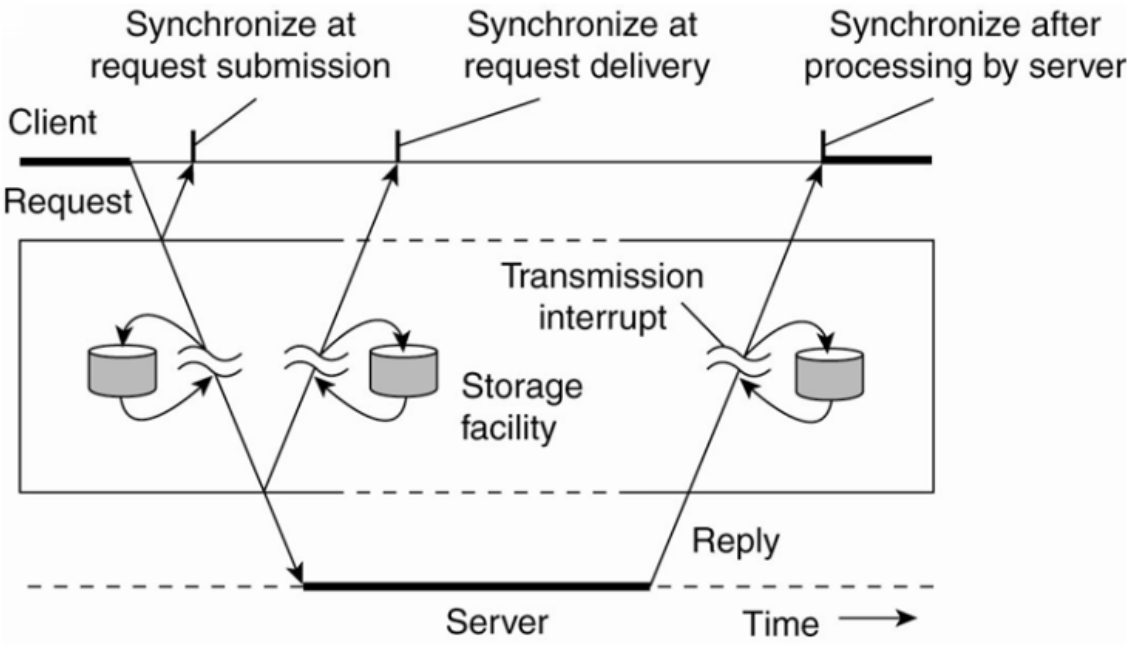
Furthermore, middleware can provide more application-specific protocols and services.

Middleware may offer different forms of communication:

- Transient vs. persistent
- Synchronous vs. asynchronous

The most popular combinations of these forms of communication are:

- Transient communication with synchronisation after processing
- Persistent communication with synchronisation at request submission

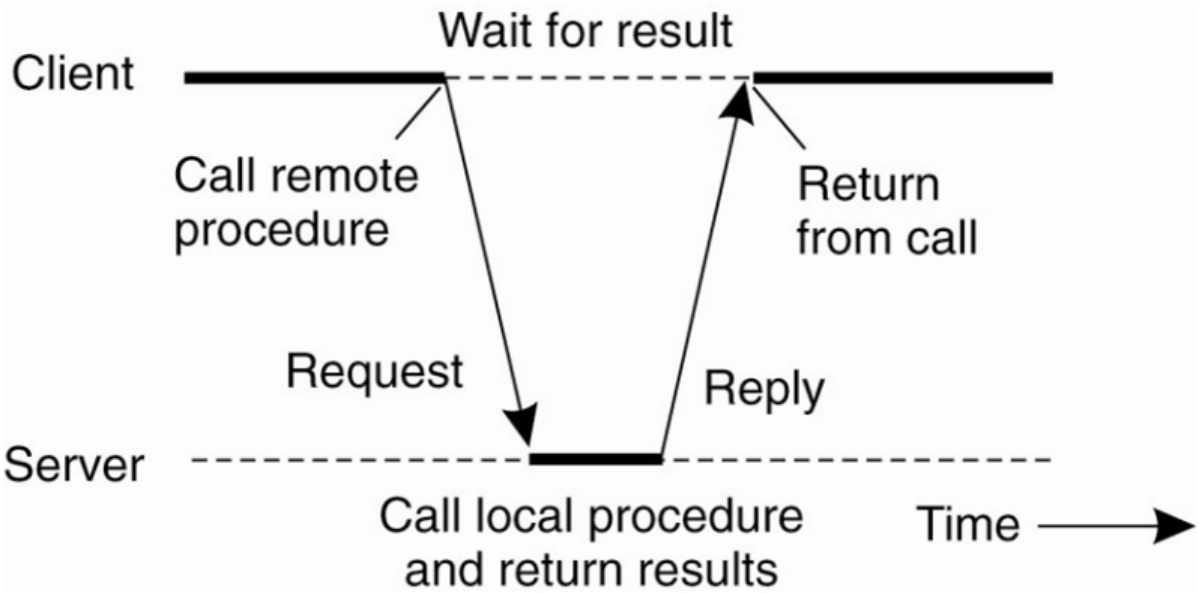


Remote Procedure Call (RPC)

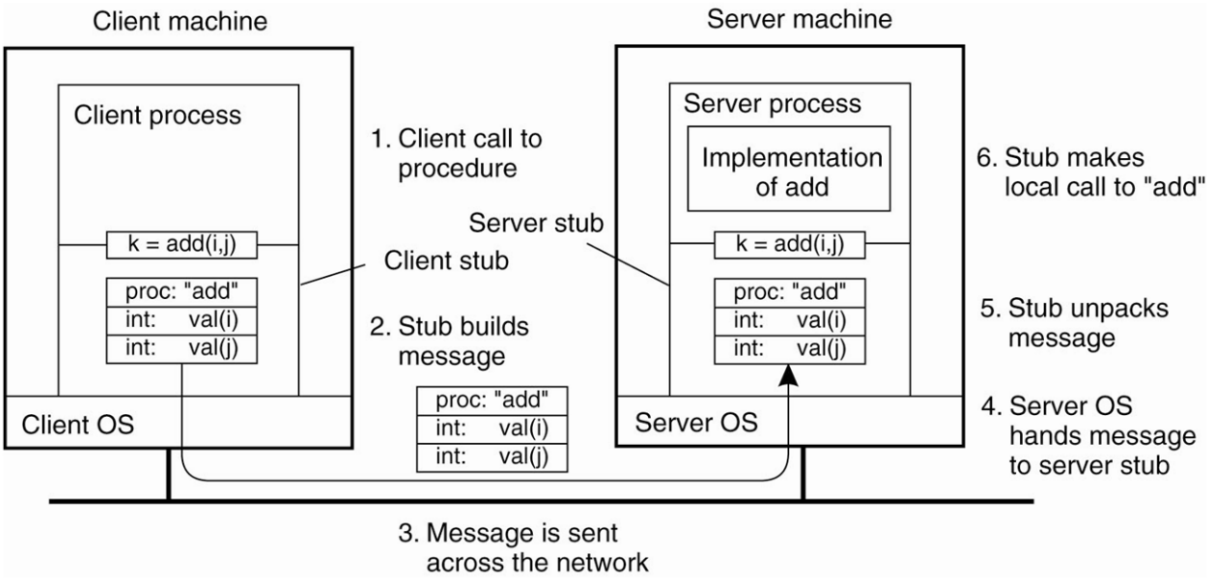
Fundamentals

A remote procedure call behaves practically the same way a *local* procedure call would: a function is called and the stack is populated with parameters and local variables of that function.

The only difference between a local and a remote procedure call is that, in the case of a RPC, the function is executed on another machine, somewhere else, and the parameters and return values are transmitted in some form via the network.



Below is a more detailed representation of how RPC works:



Parameter passing: marshalling and serialisation

Passing a parameter to a remote procedure call poses two problems:

- Structured data must be ultimately flattened in a byte stream (**serialisation** process, also known as **pickling** in the context of Object-Oriented DBMSs)
- Hosts may use different data representations (e.g. little endian vs. big endian) and proper conversions are needed (**marshalling** process)

Middleware provides **automated support** for both marshalling and serialisation. This functionality is enabled by a **language or platform independent representation** of the procedure's signature, written using an **Interface Definition Language** (*IDL*) and a **data representation** format to be used during communication.

The Interface Definition Language **raises the level of abstraction** of the service definition: it separates the service's *interface* from its *implementation.*

IDLs usually come with mappings onto target languages, such as C, Pascal, Python, and more.

The advantages of using IDLs are:

- IDLs enable the definition of services in a language-independent fashion
- Since IDLs are formal languages, they can be used to automatically generate the service interface code in the target language

Passing parameters by reference

The matter is complicated further when a remote procedure call requires a parameter to be passed **by reference** (like when passing arrays in C or objects in Java).

Many languages do not provide a notion of reference, but only of pointer, which is meaningful only within the address space of a process.

Passing parameters by reference is often not supported (i.e. in Sun's solution). Otherwise, a possibility is to use a call by value or result instead of a call by reference, although semantics are different in that case.

Furthermore, using a call by value works with arrays, but not with arbitrary data structures.

Finally, optimisations are possible if the remote procedure call is input- or output-only, not both.

RPC in practice

Sun Microsystem's RPC (also known as Open Network Computing RPC) is the **de facto standard** over the Internet: it is used at the core of NFS and many other Unix services.

oncRpc uses a data format specified by XDR (*eXternal Data Representation*) and the transport layer can be either TCP or UDP.

Parameter passing is only allowed when passing a copy of an object, so reference passing is not supported. Furthermore, there can only be one input and one output parameter per call. Security is achieved through DES.

An official standard was elaborated by the Open Group, a non-profit standardisation organisation: the **Distributed Computing Environment** (*DCE* for short).

The DCE offers several invocation semantics, such as:

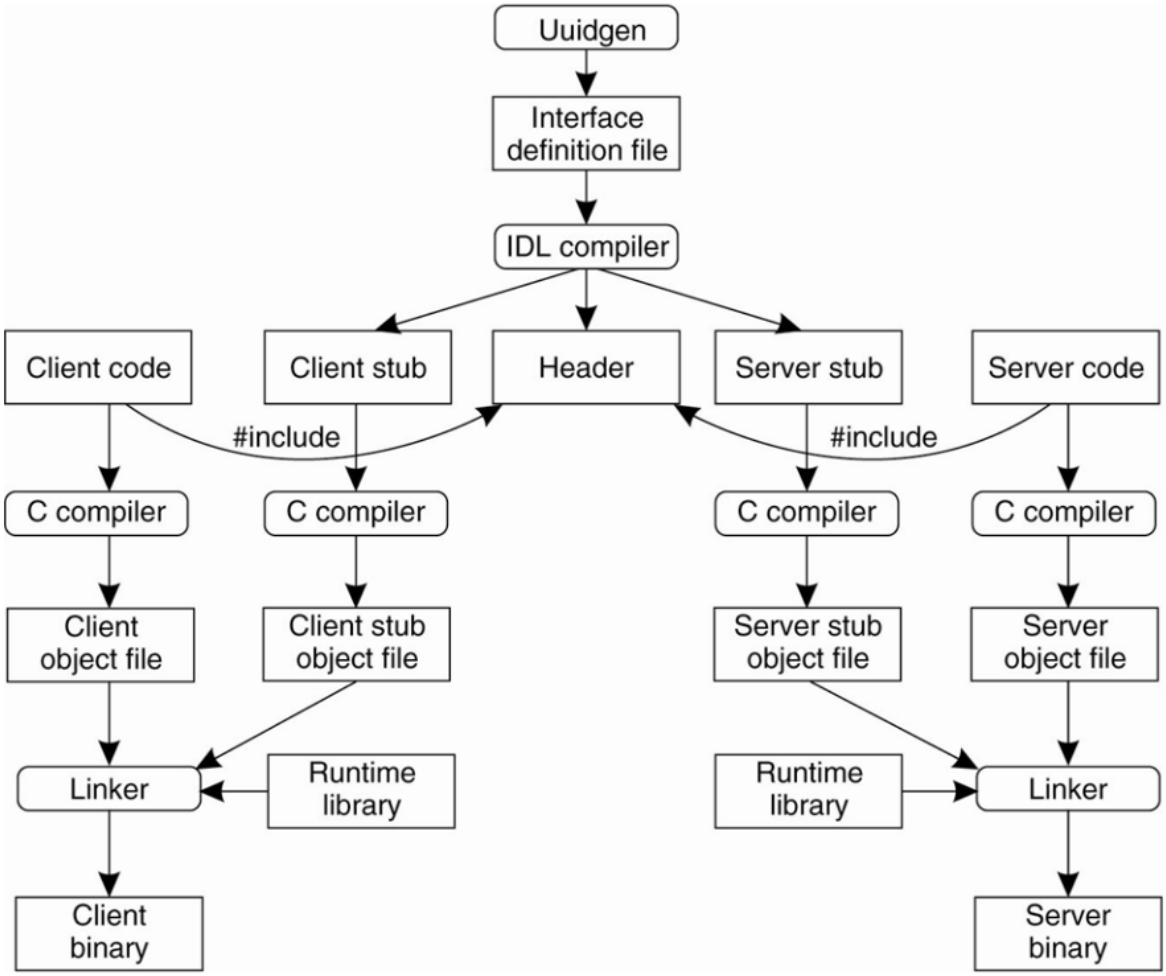
- At most once
- Idempotent
- Broadcast

Plus, several services are provided by DCE on top of RPC:

- Directory service
- Distributed time service
- Distributed file service

Kerberos is used to secure the platform.

Microsoft's DCOM and .NET remoting are based on DCE.



Binding the client to the server

> **Problem**
>
> Find out which server (or process) provides a given service

Hard-wiring information like this in the client code is **highly undesirable**.

Finding out which server provides a given service can be further divided into two problems:

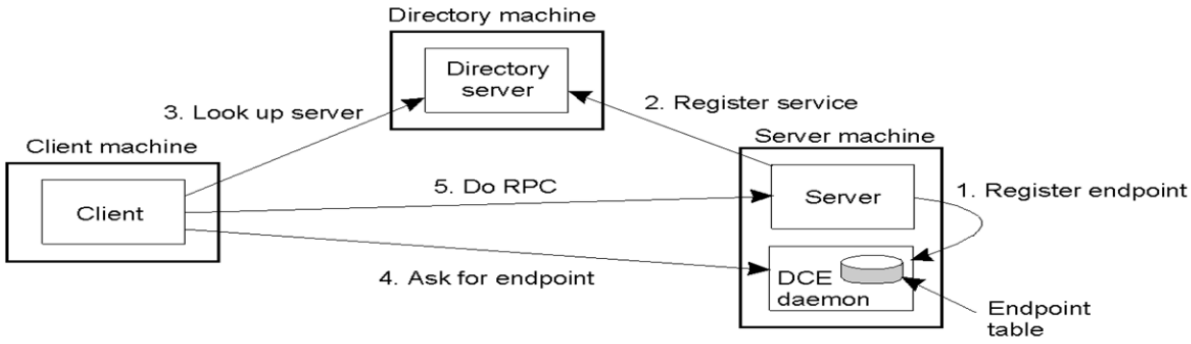
- Find out **where the server process** is
- Find out **how to establish communication** with it

In order to solve this problem, Sun decided to introduce a **daemon process** called `portmap` that **binds calls and servers or ports**: the server picks an available port and tells it to `portmap` along with the service identifier. Clients then contact a given `portmap` and request the port necessary to establish communication.

`portmap` provides its services only to local clients, which means that we still have to solve the first problem: finding the server on which the service resides.

While there are many solutions to this problem, DCE's is very similar to how `portmap` works: the DCE daemon, known as *binder daemon*, enables location transparency: the client need not know in advance where the service is, they only need to know where the **directory server** is.

In DCE, the directory service can be distributed as well, in order to improve scalability over many servers.



Dynamic activation

Problem

Server processes may remain active even in absence of requests, wasting resources

In order to solve this problem, it is possible to introduce **another server daemon** that:

- Forks the process to serve the request
- Redirects the request if the process is already active

This obviously means that the first request is served less efficiently.

In oncRpc, this process is accomplished through the `inetd` daemon: the mapping between the requested service and the server process is stored in a configuration file in `/etc/services`.

Lightweight RPC

It is natural to use the same primitives for inter-process communication, regardless of distribution, but using conventional RPC would lead to wasted resources, since there's no need to use TCP or UDP when communicating on a single machine.

Lightweight RPC solves this problem by passing messages using local facilities: communication exploits a private shared memory region.

Lightweight RPC is invoked as follows:

1. The client stub copies the parameters on the shared stack and then performs a system call
2. The kernel does a context switch to execute the procedure in the server
3. Results are copied on the stack and another system call and context switch brings execution back to the client

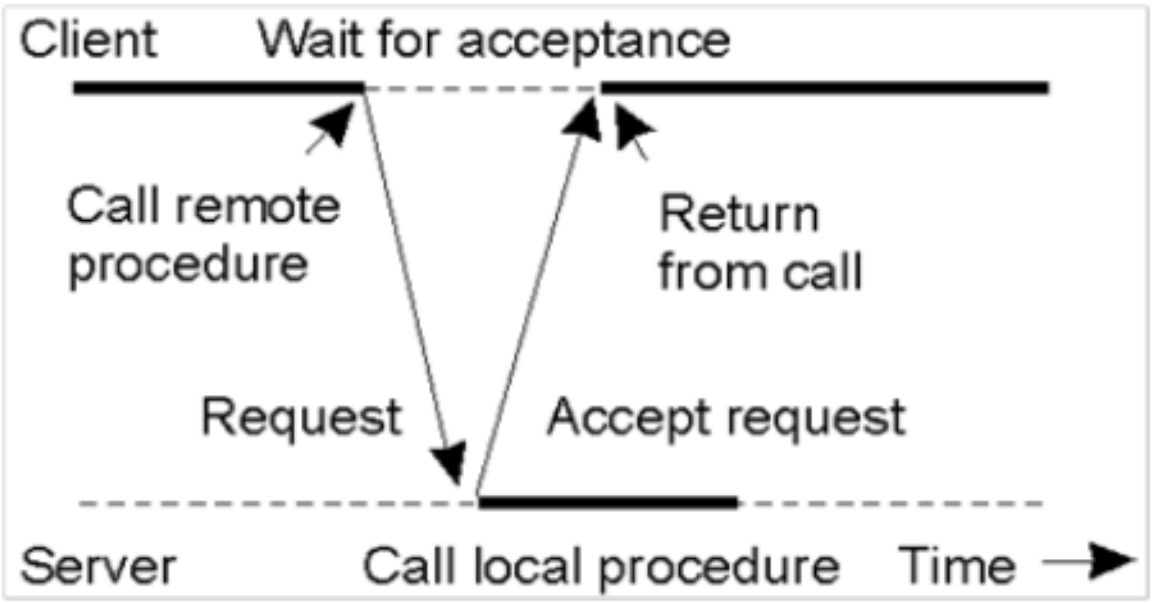
The advantages of using lightweight RPC are that less threads and processes are used and only one parameter copy is needed instead of the normal four a standard RPC would need.

Similar concepts are used in practice in DCOM and .NET.

Asynchronous RPC

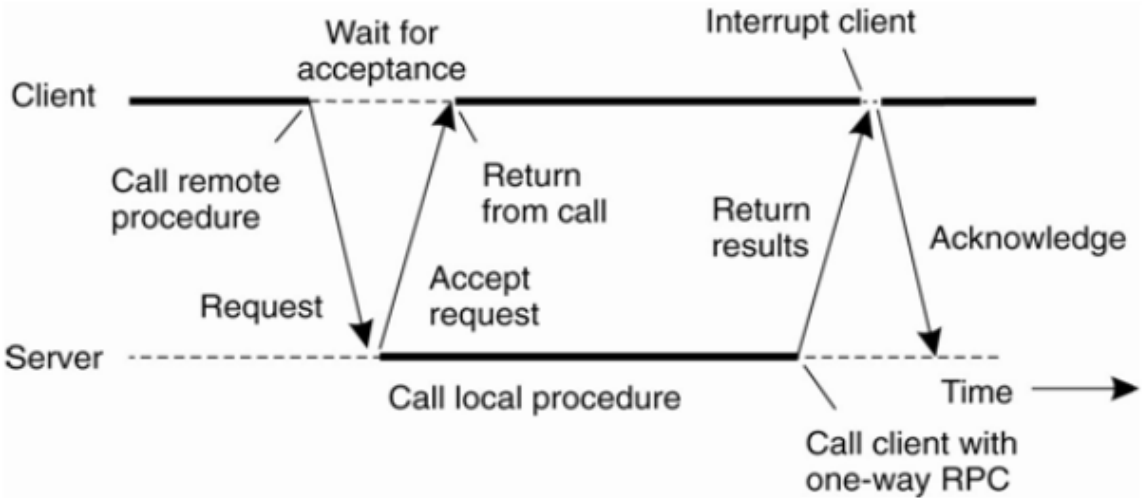
Many variants of RPC are **asynchronous**, since synchronous remote procedure calls can potentially waste client resources.

If no result is needed, execution can resume after an acknowledgement is received from the server. There is no need to wait for the execution of the remote call. This is known as **One-way RPC** and it returns immediately.



asynchronous
(one possible)

If a return value is needed, then the callee may asynchronously invoke the caller back. Alternatively, invocation may return immediately with a **promise** (or **future**), which can be polled by the client later in order to get the needed results. This is known as **deferred synchronous** RPC.



deferred synchronous

Batched vs. queued RPC

Sun's implementation of RPC includes the ability to perform a **batched** RPC: RPCs that do not require a result are buffered on the client and are sent together when a non-batched call is requested (or when a timeout expires). This enables yet another form of asynchronous RPC.

A similar concept can be used to deal with mobility, as seen in the Rover toolkit by MIT: if a mobile host is disconnected between sending the request and receiving its reply, the server periodically tries to contact the mobile host and deliver the reply. Requests and replies can also come through on different channels.

Depending on network conditions and application requirements, the network scheduler may decide to:

- Send requests in batches
- Compress the data
- Reorder requests and replies in a non-FIFO order (e.g. to suit application-specified priorities)

Remote Method Invocation (RMI)

Remote method invocation follows the same principles as RPC, but with different programming constructs: the goal of RMI is to obtain the advantages of Object-Oriented Programming (OOP) in the distributed setting.

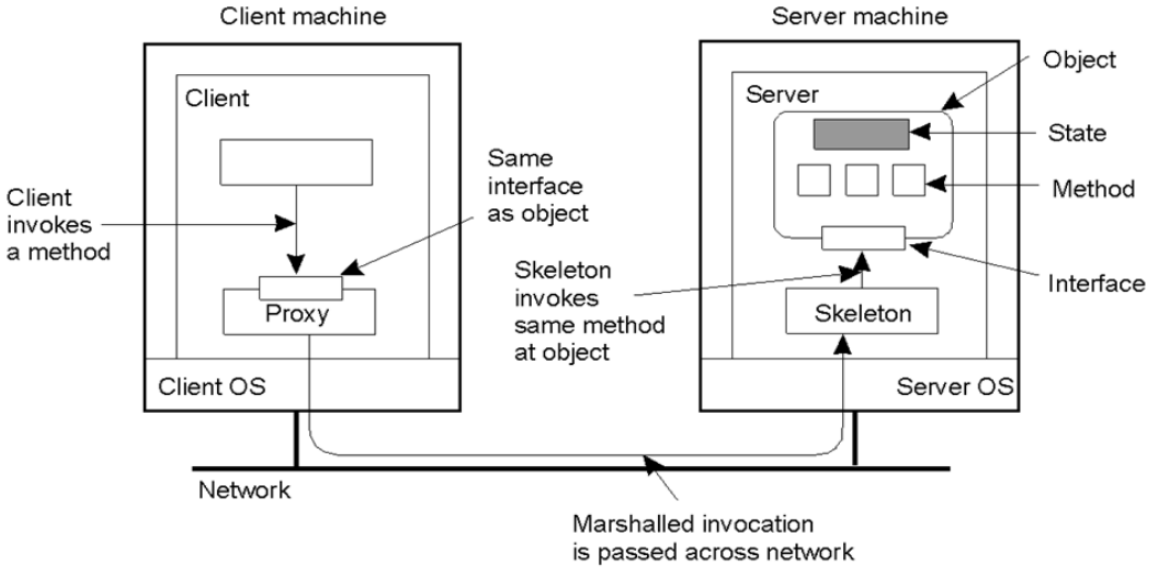
RMI has one important difference when compared to RPC: remote object references **can be passed around**.

Often times, RMI frameworks are built on top of RPC layers.

Interface definition language

In RPC, the IDL separates the interface from its implementation in order to handle platform and language heterogeneity. Such separation is one of the **basic OO principles**, so it becomes natural to place the object interface on one host and the implementation on another.

IDLs for distributed objects are **much richer** than the ones used for RPC, since they can account for inheritance, exception handling, and more.



Remote method invocation in practice

There are many examples of practical RMI frameworks. One of the most popular is Java RMI, which has the following characteristics:

- It is based on a single language, Java, and platform, the Java Virtual Machine (JVM)
- It easily supports passing parameters by reference or by value, even with complex objects
- It supports the "code on-demand" framework

Another very popular RMI framework is OMG CORBA, a multilanguage, multiplatform RMI framework which also supports passing parameters by reference.

Message-oriented communication

RPC and RMI foster a synchronous model, since they are a natural programming abstraction, but they have some downsides:

- They support point-to-point interaction only
- Synchronous communication is expensive
- The intrinsically tight coupling between caller and callee leads to **rigid** architectures

In order to solve these problems, **message-oriented communication** is many times preferred, since:

- It's centred around the simpler notion of one-way message or event
- It is usually asynchronous
- It often supports persistent communcation
- It often supports multi-point interaction
- It brings more decoupling among components

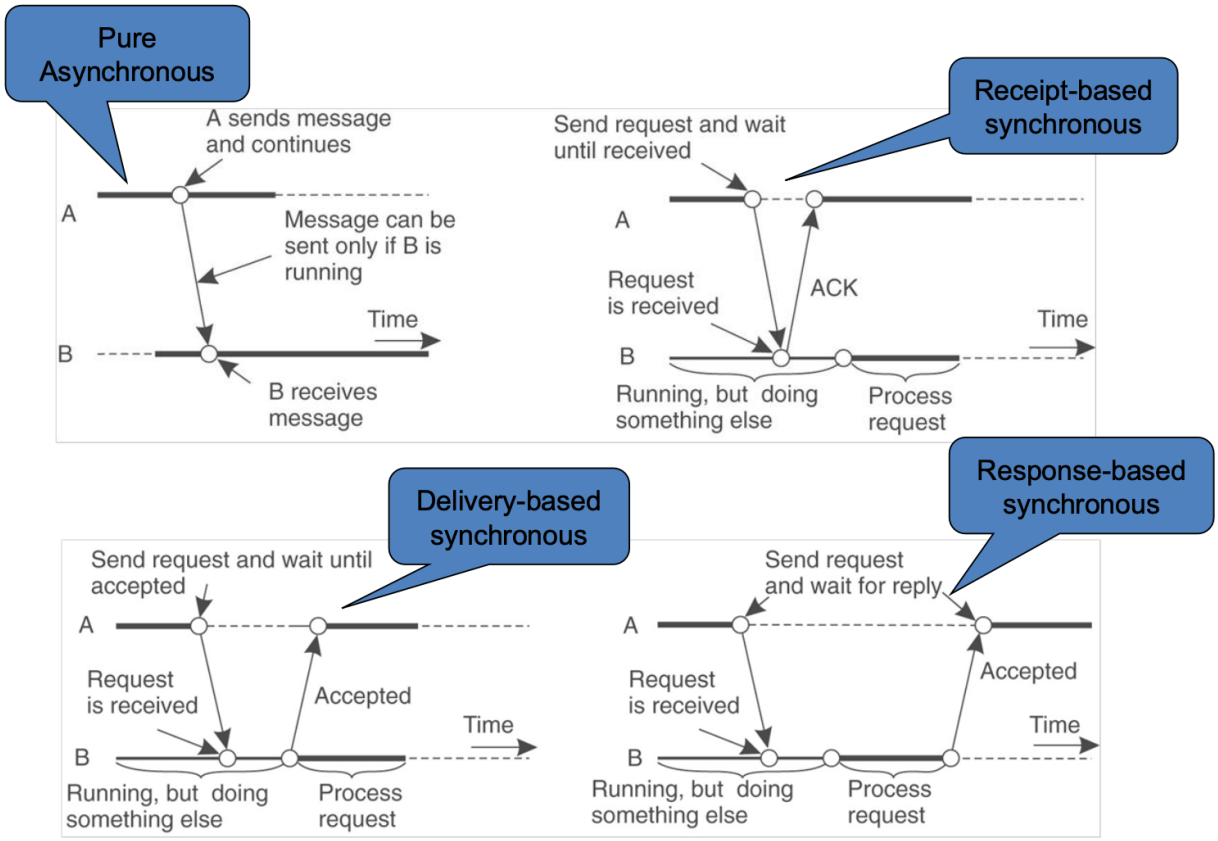
Types of communcation

In message-oriented communication, we still get the types of communication cited before:

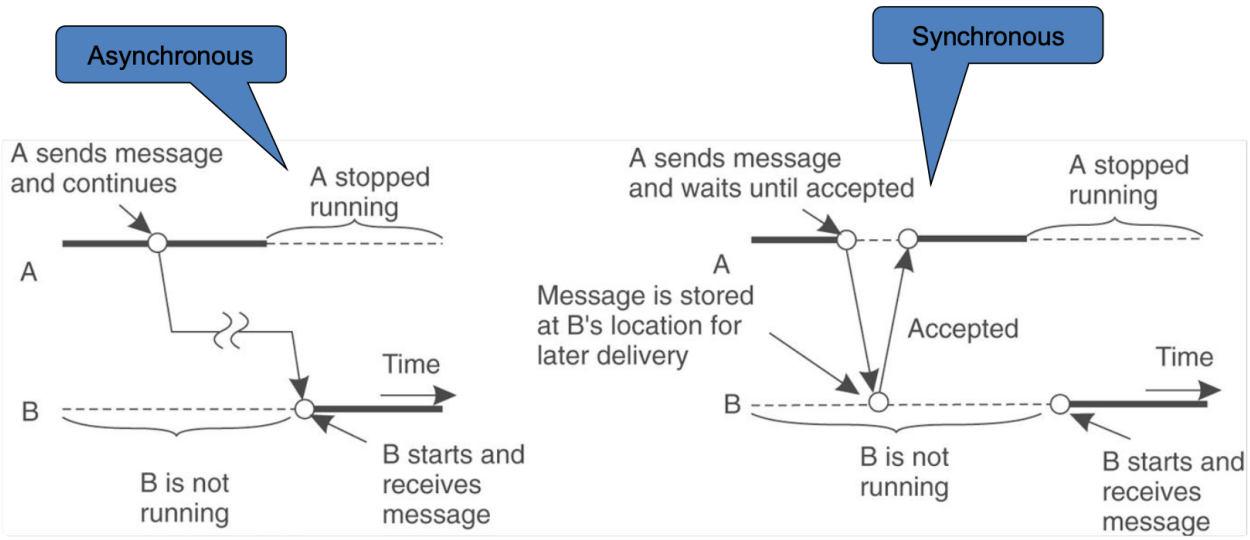
- **Synchronous** communication: the sender is blocked until the recipient has stored the message
- **Asynchronous** communication: the sender continues immediately after sending the message
- **Transient** communication: sender and receiver must both be running for the message to be delivered
- **Persistent** communication: the message is stored in the communication system until it can be delivered

Several alternatives and combinations of these types of communication are provided in real world scenarios. Below, you can see some of them:

Transient communication



Persistent communication



Reference model

The most straightforward form of message oriented communication is **message passing**, which is typically directly mapped on or provided by the underlying network OS functionality, like sockets.

A kind of middleware can also provide another form of message passing called Message Passing Interface (*MPI*).

Message queuing and publish/subscribe are two different models provided at the middleware layer by several communication servers through what is called *overlay network*.

From network protocols to communication

Unicast TCP and UDP are well known network protocols, but how can a programmer take advantage of them for their own projects?

Berkeley sockets are the answer. Berkeley sockets first appeared in Unix BSD in 1982 and are now available everywhere. They provide a common abstraction for inter-process communication: they allow both connection-oriented (stream, i.e. TCP) and connectionless (datagram, i.e. UDP) communication.

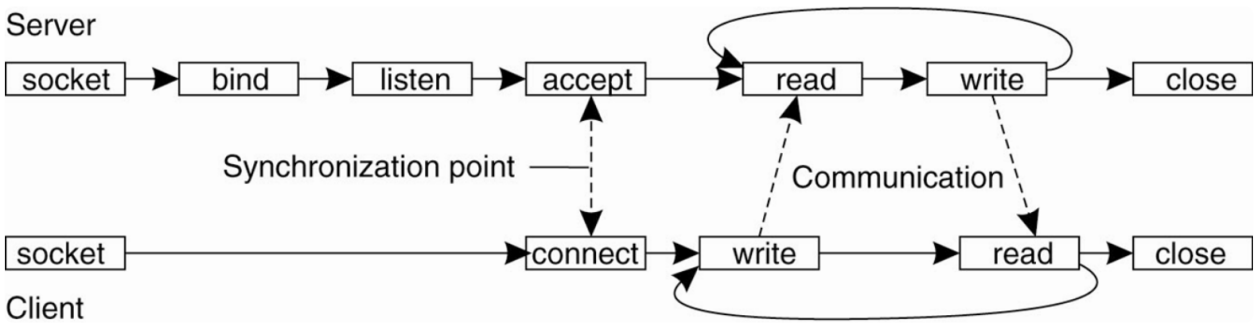
Stream sockets: fundamentals

When using connection-oriented sockets, the server accepts connections on an open **port** and the client connects to the server through that port.

Each connected socket is uniquely identified by four numbers:

- The IP address of the server
- Its incoming port
- The IP address of the client
- Its outgoing port

In C, the basic workflow for connection-oriented sockets is the following:

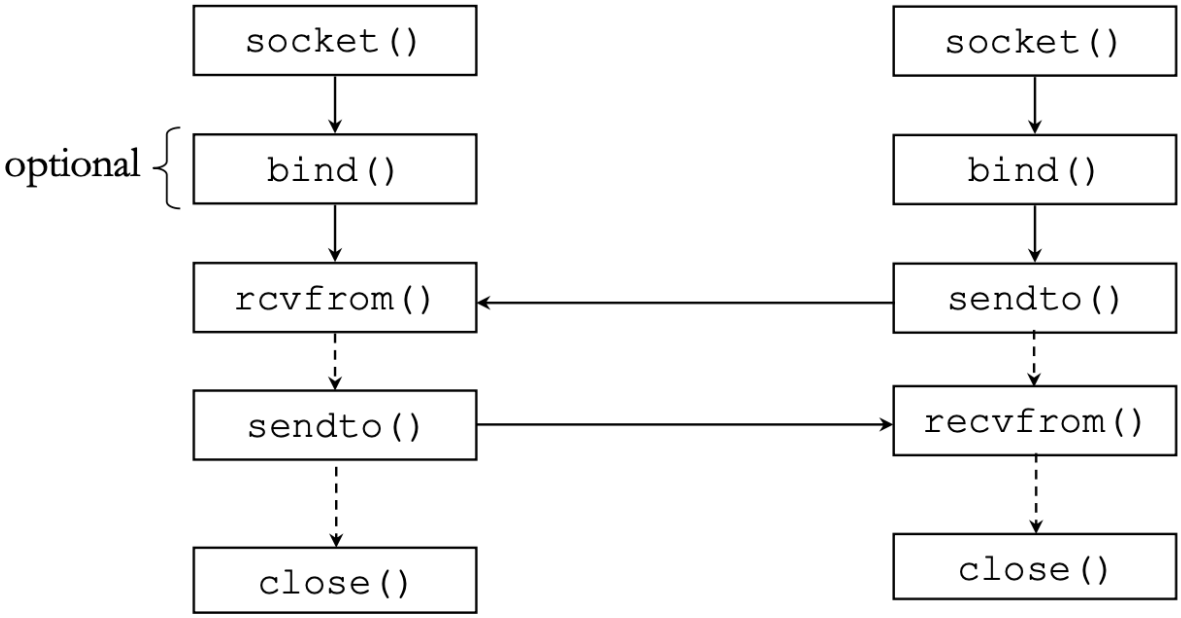


Datagram sockets: fundamentals

Clients and servers use the same approach to send and receive datagrams: both create a socket bound to a port and use it to send and receive messages.

When using datagram communication, there is **no connection** and the same socket can be used to send and receive datagrams to and from multiple hosts.

The workflow to use datagram sockets in C, as an example, is the following:



Multicast sockets

IP multicast is a network protocol used to efficiently deliver UDP datagrams to multiple recipients.

IP reserves a class D address space, from 224.0.0.0 to 239.255.255.255, to multicast groups.

The socket API for multicast communication is similar to that for datagram communication: the components interested in receiving multicast datagrams addressed to a specific group must **join the group** using the `setsockopt` call (unless a group is open: in that case, it is not necessary to become a member of the group to receive messages). As usual, it is also necessary to specify a port.

Note

> Most routers are configured not to route multicast packets outside of the LAN.

Message Passing Interface (MPI): fundamentals

The Message Passing Interface (MPI in short) is a platform-independent answer to the limitations of sockets, which are low level and protocol independent.

In high performance networks, higher level primitives are needed for asynchronous transient communication and there may be a need to provide different services other than read and write.

In MPI, communication takes place within a known group of processes. Each process within a group is assigned a local ID: the pair `(groupId, processId)` represents a source or destination address.

Messages can also be sent in broadcast to the entire group through primitives such as `MPI_Bcast`, `MPI_Reduce`, `MPI_Scatter`, `MPI_Gather`.

MPI has **no support for fault tolerance**: crashes are supposed to be fatal.

Here's a list of the main MPI primitives:

Primitive	Description
`MPI_Bsend`	Append outgoing message to a local send buffer
`MPI_Send`	Send a message and wait until copied to local or remote buffer
`MPI_Ssend`	Send a message and wait until recipient starts
`MPI_Sendrecv`	Send a message and wait for a reply
`MPI_Isend`	Pass a reference to an outgoing message and continue
`MPI_Issend`	Pass a reference to an outgoing message and wait until recipient starts
`MPI_Recv`	Receive a message and block if there are no messages
`MPI_Irecv`	Check if there's an incoming message but do not block

Message queuing

Point-to-point persistent asynchronous communication typically only guarantees the eventual insertion of the message in the recipient queue, but it does not guarantee that the recipient will actually read the message. Because of this,

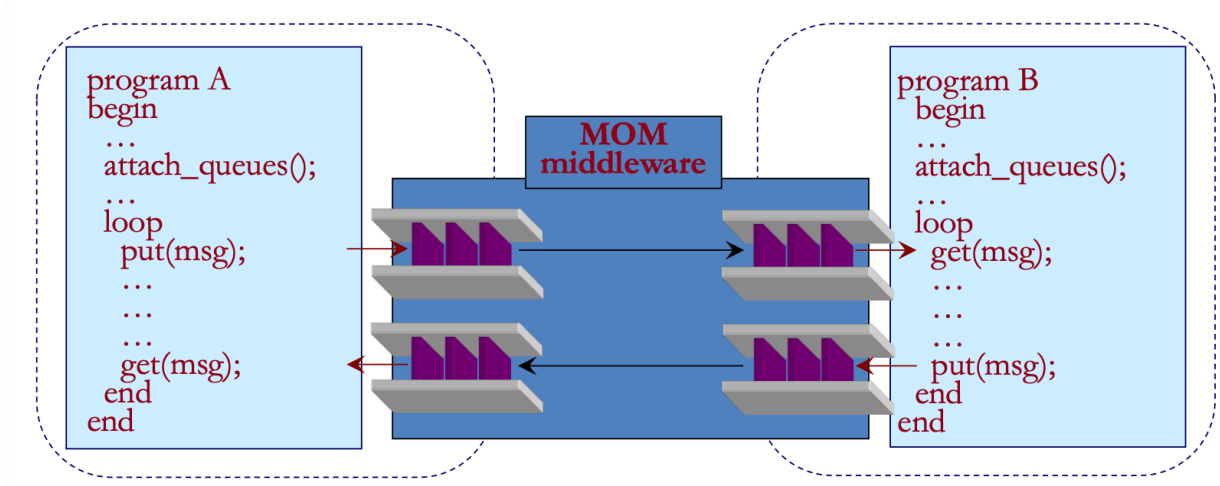
communication is decoupled in time and space and it can be regarded as a *generalisation of e-mails*.

Message queuing is an intrinsically peer-to-peer architecture, in which each component holds an input queue and an output queue.

This type of communication architecture is found in many commercial systems, like the IBM's WebSphere MQ, Microsoft Message Queues and Java Message Service.

The basic communication primitives in message queuing-based architectures are:

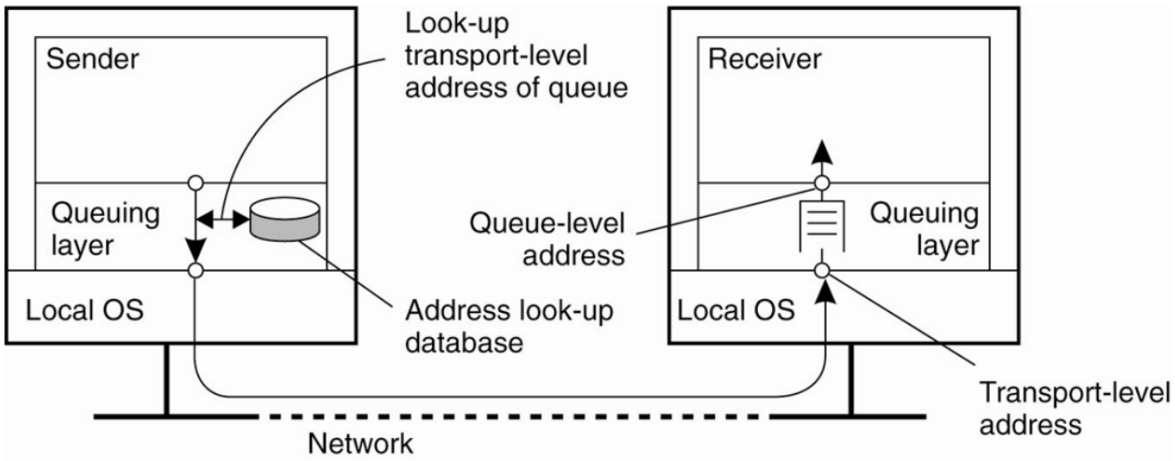
Primitive	Description
`Put`	Append a message to a specified queue
`Get`	Block until the specified queue is nonempty, then remove the first message
`Poll`	Check a specified queue for messages, then remove the first, but never block the queue
`Notify`	Install a handler to be called when a message is put into a specified queue



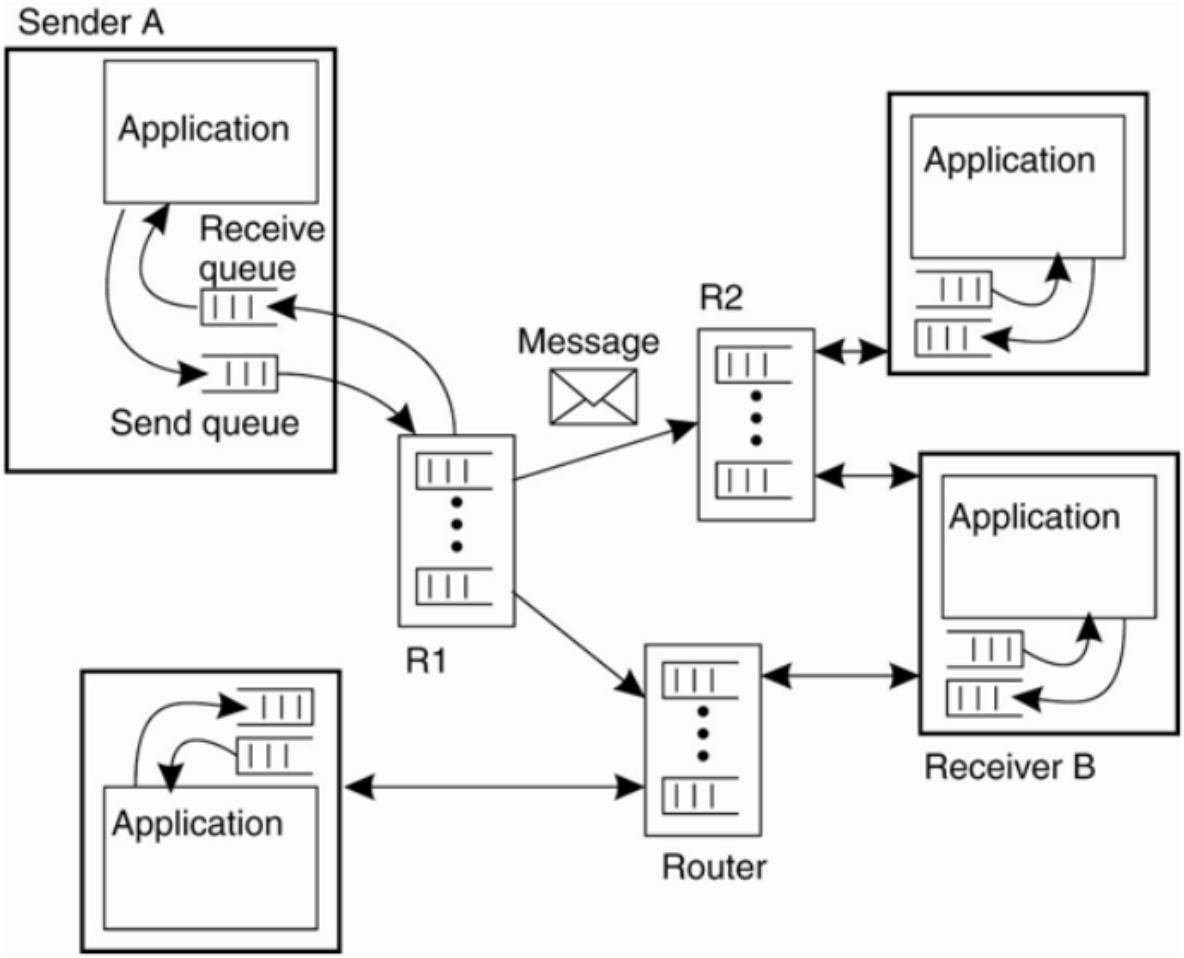
Message queuing is especially useful in client-server architectures that don't need to be synchronous: clients send requests to the server's queue and then the server **asynchronously** fetches the request, processes it and returns the result to the client's queue.

Thanks to persistence and asynchronicity, clients **do not need to remain connected**. Furthermore, queue sharing simplifies server load balancing.

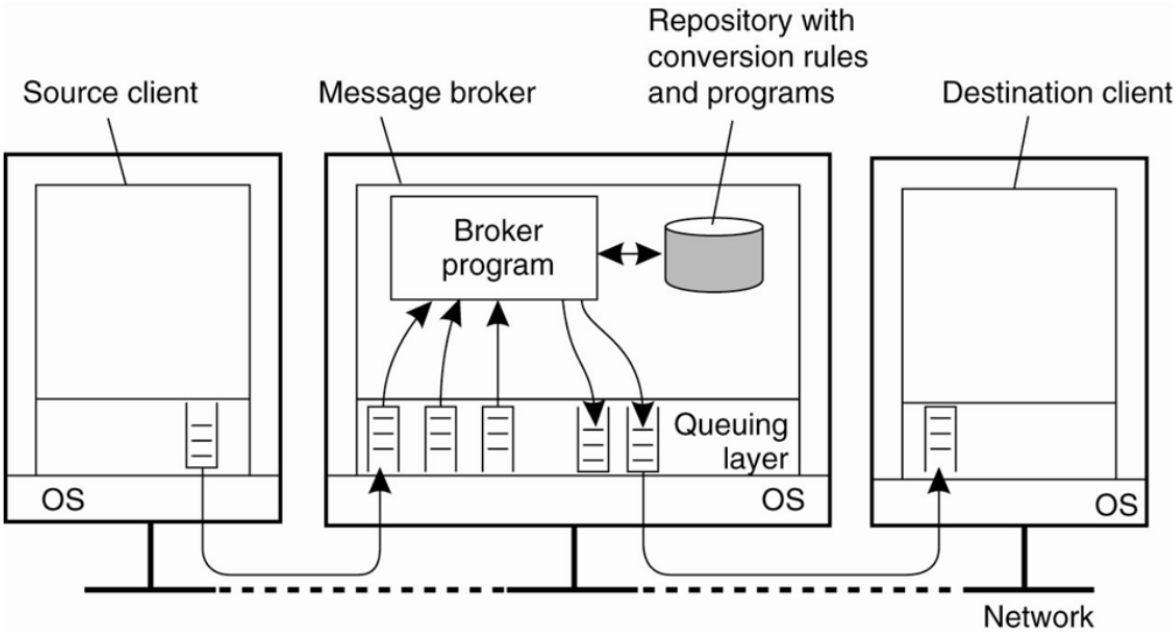
Message queuing also has some architectural issues, though. For example, queues are identified through **symbolic names**, which means that a lookup service must be put in place to find the queues in the first place. Message queues usually use pre-deployed static names.



Furthermore, queues are manipulated by queue managers, which can be local and/or remote and act as relays (or *applicative routers*). Relays are often organised in an overlay network, in which messages are routed using application-level criteria and relying on partial knowledge of the network. This improves fault tolerance and provides applications with multi-point without the need for IP-level multicast.



Finally, message brokers provide application-level gateways supporting message conversion, which is useful when integrating sub-systems.



Publish-subscribe

In the publish-subscribe paradigm, application components can publish asynchronous event notifications and declare their interest in event classes by issuing a subscription. This paradigm uses an extremely simple API, composed of two primitives only: `publish` and `subscribe`. Event notifications are but simple messages.

Subscriptions are collected by an **event dispatcher** component, responsible for routing events to all matching subscribers. An event dispatcher can be either centralised or distributed.

In the publish-subscribe architecture, communication is:

- Transiently asynchronous
- Implicit
- Multipoint

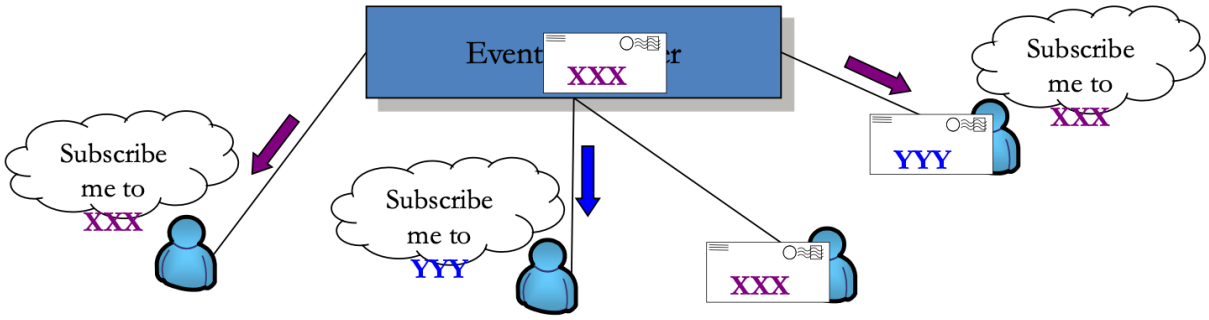
Furthermore, this architecture provides a high degree of decoupling among components, which makes it easy to add and remove components and is appropriate for dynamic environments.

In the publish-subscribe architecture, to express the publish and subscribe events, a **subscription language** is used. Its expressiveness allows to distinguish between:

- **Subject-based** (or *topic-based*) subscriptions: applications can subscribe to specific subjects, which are determined beforehand (e.g. "subscribe to all events about world news")
- **Content-based** subscriptions: subscriptions contain expressions, called *event filters*, which allow clients to filter events based on their content. In this case, a single event may match multiple subscriptions (e.g. "subscribe to all events about a distributed systems class with date greater than 2004-11-16 and held in classroom D04")

These two types of subscriptions can be combined. However, the more expressive a subscription language is, the more complex its implementation becomes.

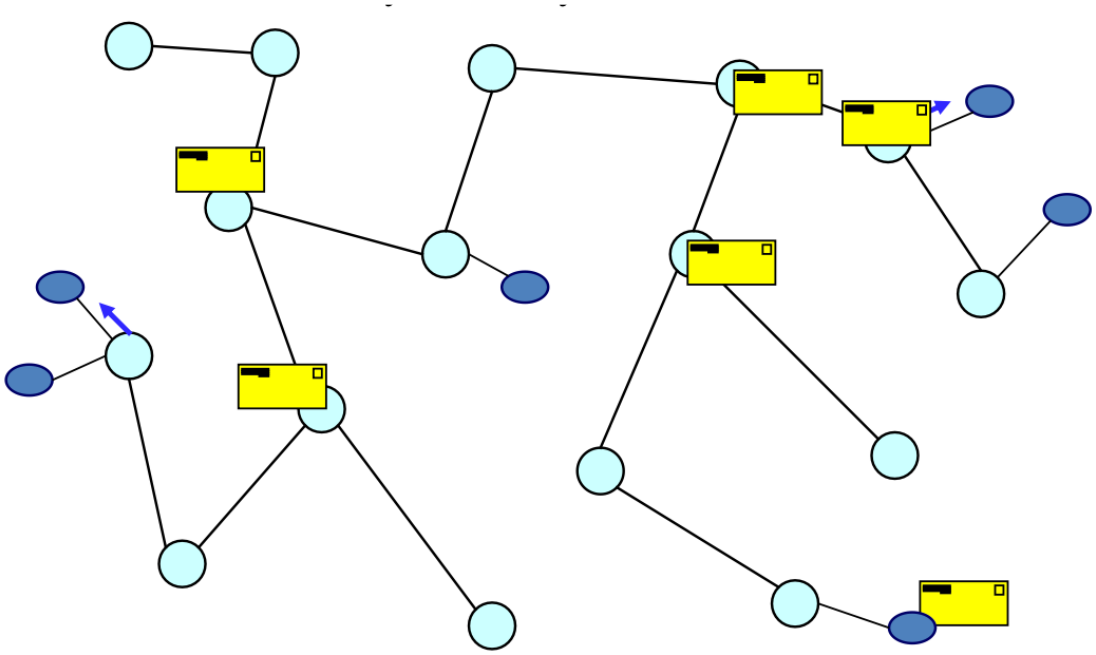
The other fundamental component of a publish-subscribe architecture is the **event dispatcher**, a special component in charge of collecting subscriptions and routing event notifications based on such subscriptions.



An event dispatcher can be:

- **Centralised**: a single component is in charge of collecting subscriptions and forward messages to subscribers
- **Distributed**: a set of *message brokers* organised in an *overlay network* cooperate to collect subscriptions and route messages. The topology of the overlay network and the routing strategy adopted by the system may vary

Depending on the topology of the overlay network, messages can be forwarded in different ways. When using an **acyclic graph**, every broker stores subscriptions coming from directly connected clients only. Messages are forwarded from broker to broker and only delivered to clients that are subscribed to that type of event.



Specifically:

- Every broker forwards subscriptions to the others
- Subscriptions are never sent twice over the same link
- Messages follow the routes laid by subscriptions

Each time a broker receives a message, it must match it against the list of received filters in order to determine the list of recipients. The efficiency of this process may vary, depending on the complexity of the subscription language and the implemented forwarding algorithm.

One possible forwarding algorithm is **hierarchical forwarding**, which assumes that the overlay graph is a rooted tree. In hierarchical forwarding, both messages and subscriptions are forwarded by brokers towards the root of the tree. Messages flow *downwards* only if a matching subscription had been received along that route.

When working with **cyclic graphs** instead, it's possible to use a Distributed Hash Table (DHT in short) in order to forward messages correctly.

A DHT organises nodes in a structured overlay, allowing for efficient routing towards the node having the smallest possible ID that is greater or equal to the ID specified by the messages.

To subscribe for messages with a given subject `s`, a node must follow this procedure:

- Calculate the hash `H` of the subject `s`
- Use the distributed hash table to route towards the node which is the successor of the node with ID `H`
- While flowing towards the successor, leave routing information to return messages back

In order to publish messages of a given subject `s` instead:

- Calculate the hash `H` of the subject `s`
- Use the distributed hash table to route towards the successor of the node with ID `H`
- While flowing towards that node, follow back routes used to share subscriptions

For content-based routing, there are many different forwarding strategies, like:

- Per-source forwarding (PSF)
- Improved per-source forwarding (iPSF)
- Per-receiver forwarding (PRF)

There are also different strategies to build paths, for instance:

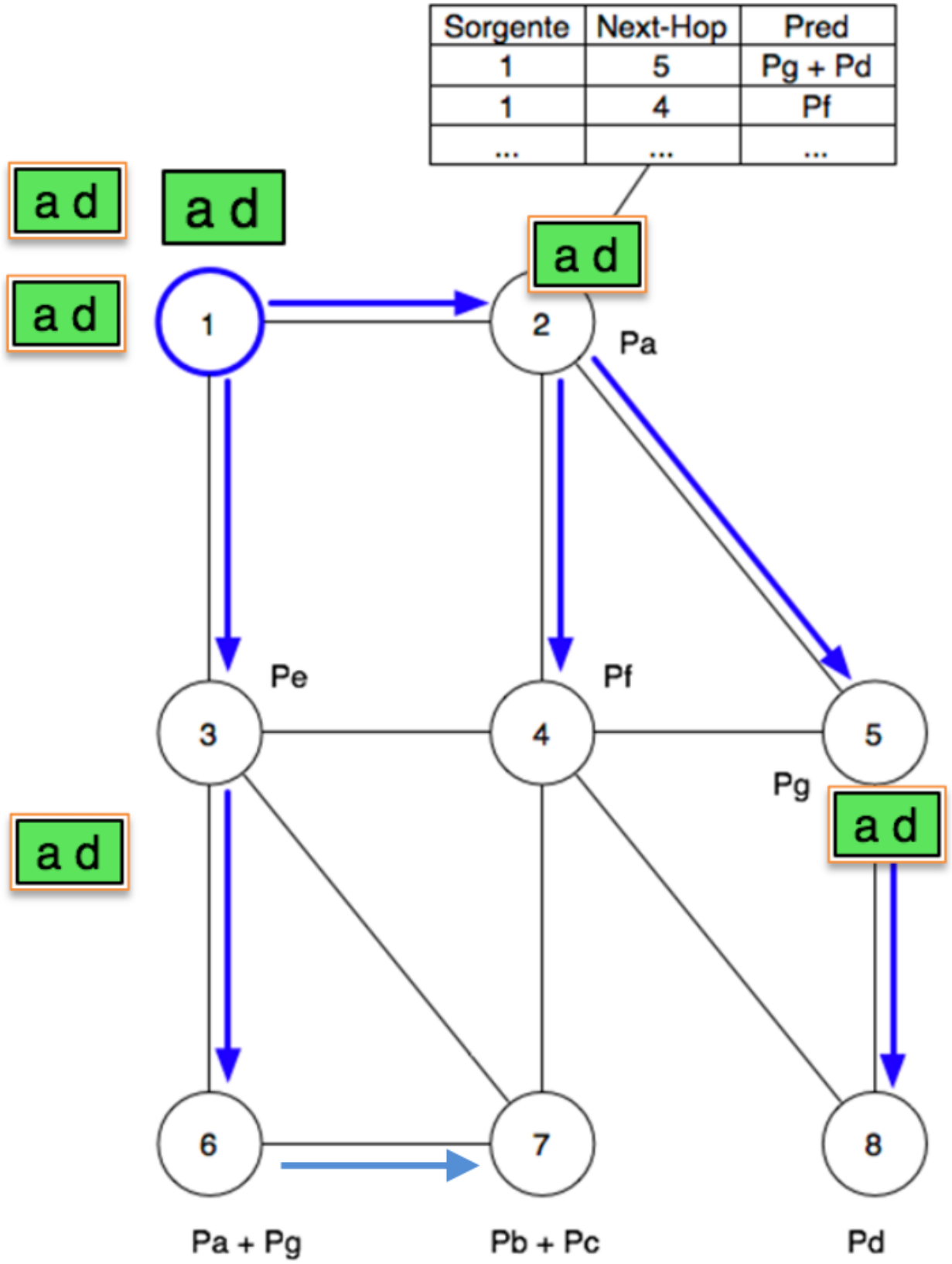
- Distance Vector (DV)
- Link-State (LS)

Once paths have been computed, every node populates forwarding tables to save the routing information.

Per-source forwarding

In PSF, every source defines a shortest path tree (SPT) and the forwarding table keeps the routing information organised per-source: for each source `v`, the children in the SPT are associated with `v`, while for each child `u`, a predicate which joins the predicates of all the nodes reachable from `u` along the SPT.

⚡ Non ho capito



Improved per-source forwarding

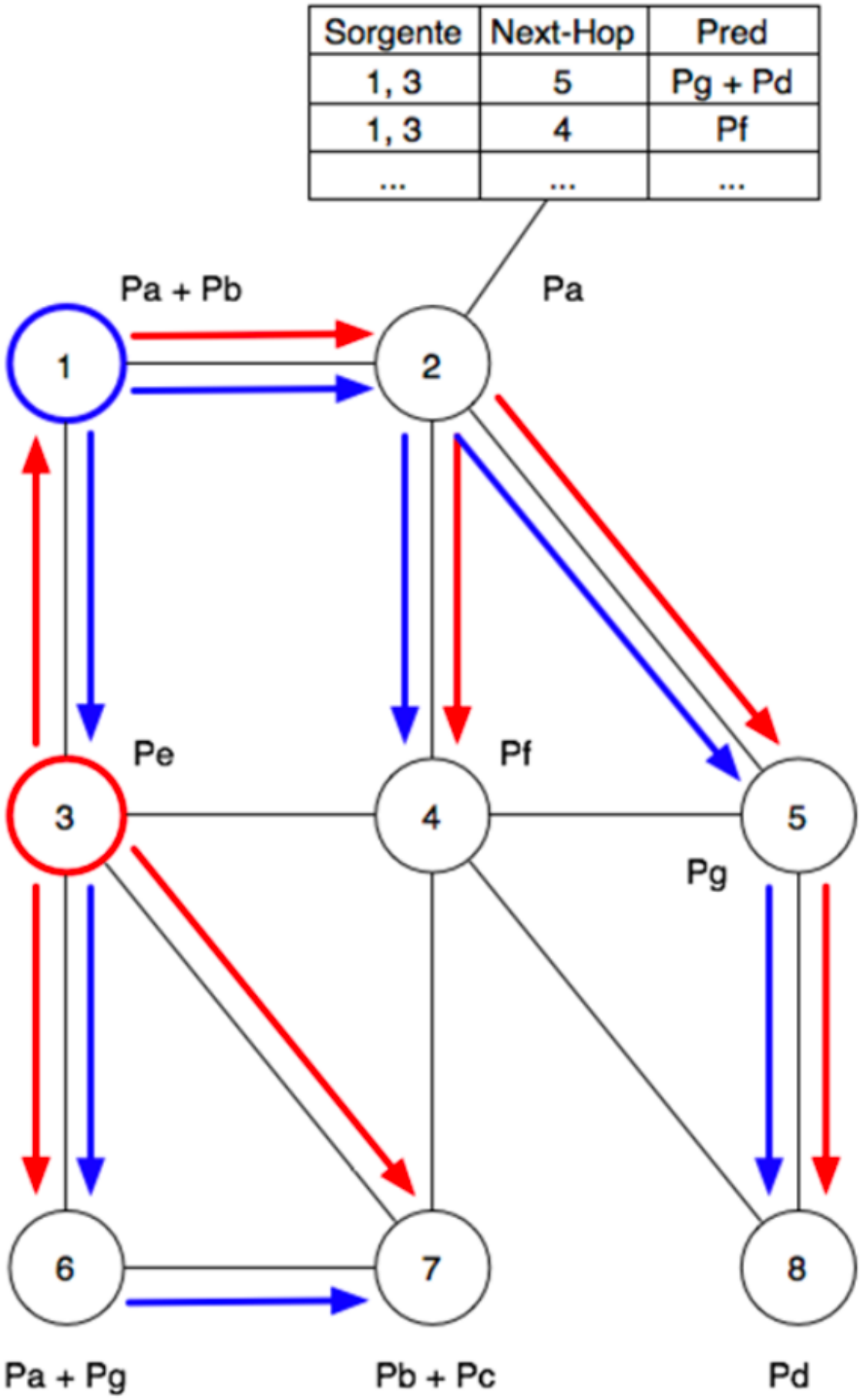
This algorithm is the same as PSF, but leverages the concept of **indistinguishable sources**:

> ⓘ **Indistinguishable nodes**

Two sources A, B with SPTs $T(A), T(B)$ are **indistinguishable** from a node n if n has the same children for $T(A)$ and $T(B)$ and reaches the same nodes along those children.

This concept brings several advantages:

- Smaller forwarding tables
- Easier to compute the paths for the forwarding tables

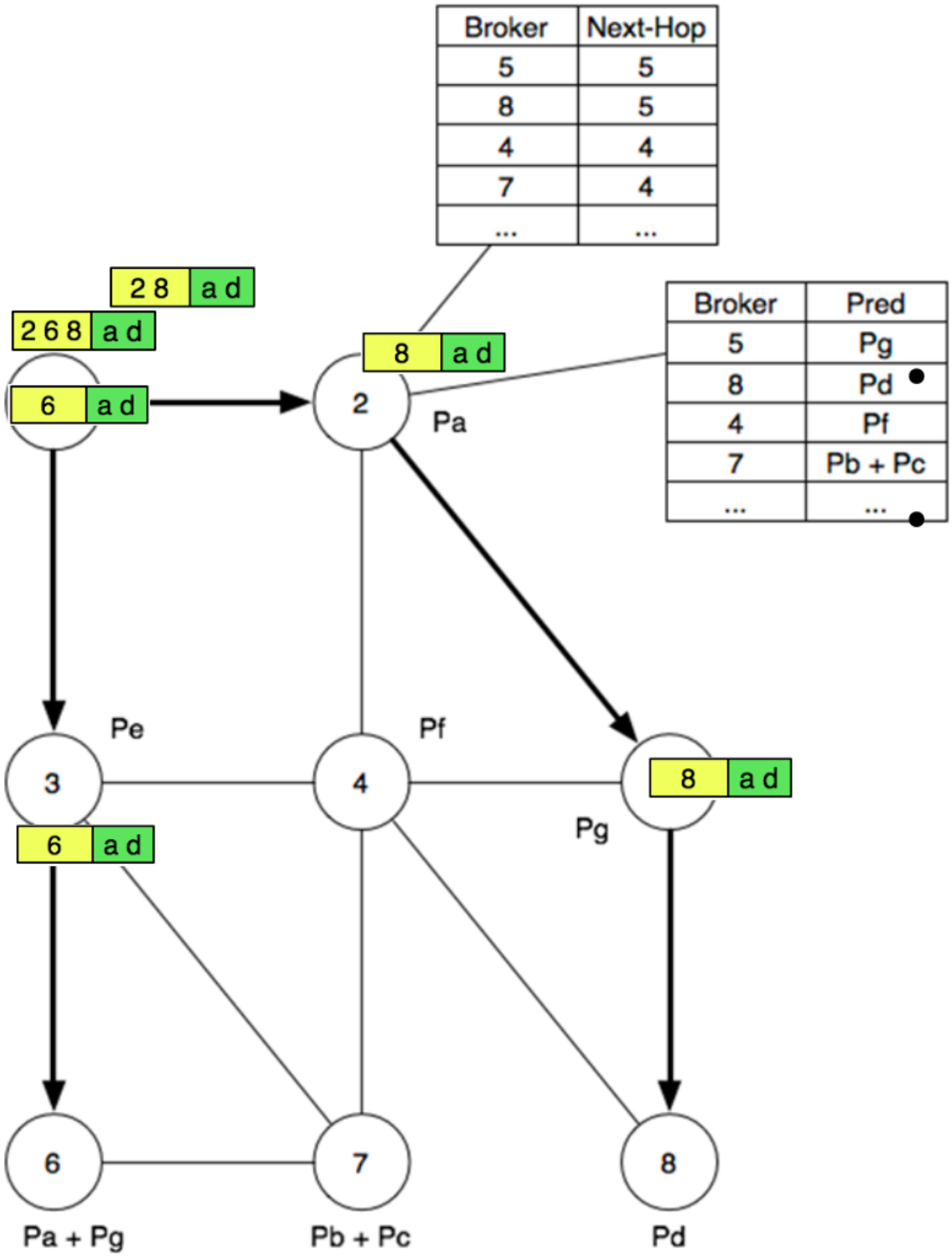


Per-receiver forwarding

In PRF, the source of a message calculates the set of receivers and adds them to the header of the message. At each hop, the set of recipients is partitioned.

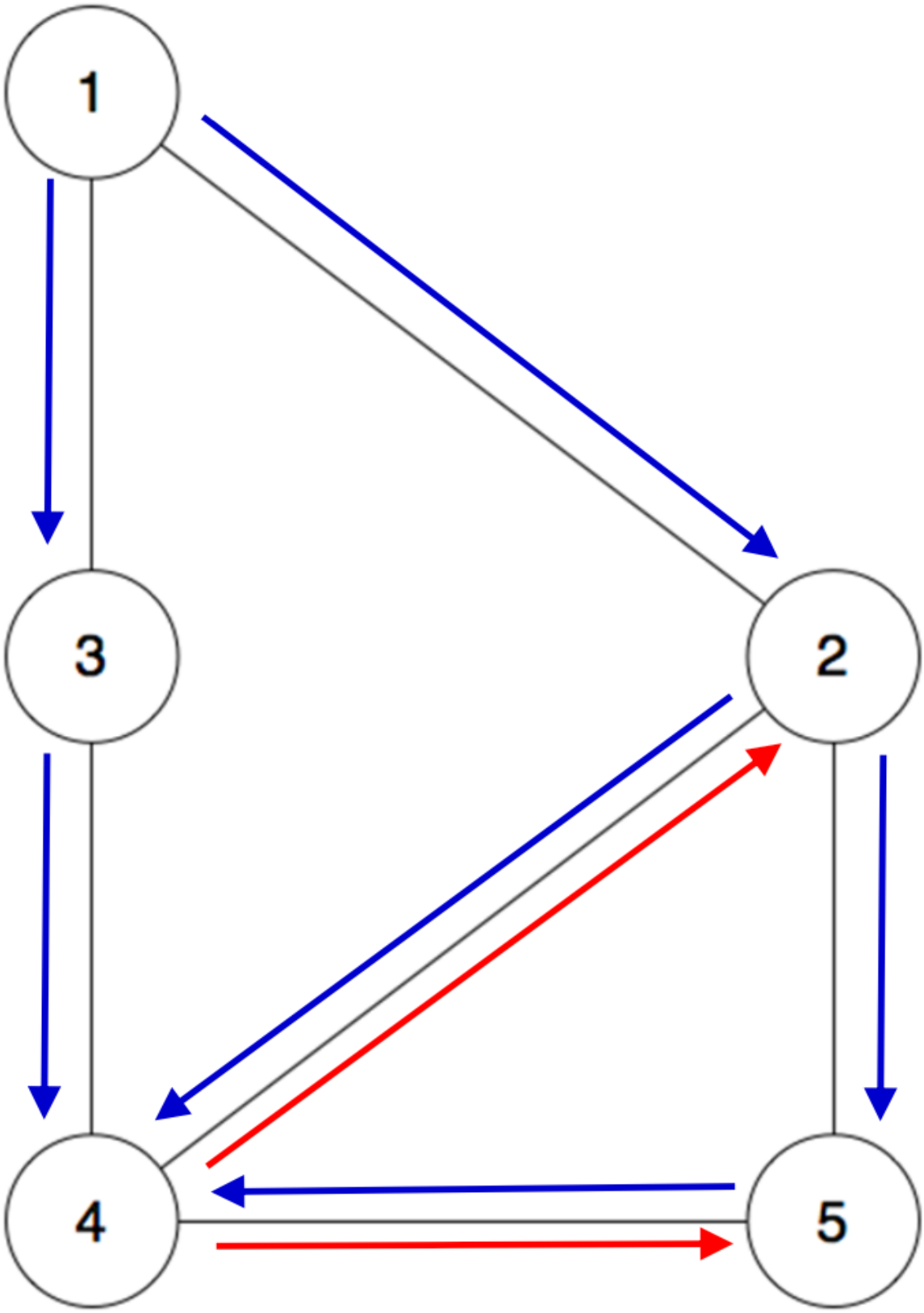
PRF uses two different tables:

- The unicast routing table
- A forwarding table with the predicate for each node in the network



The Distance Vector algorithm builds minimum latency SPTs through request and reply packets (respectively, `config` and `config response`).

With the DV algorithm, every node acquires a local view of the network.



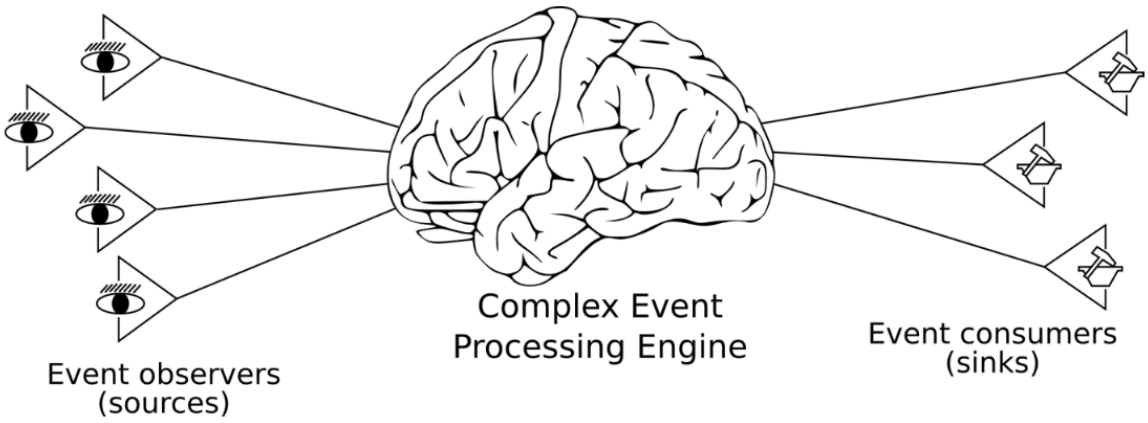
The Link-State algorithm allows to build SPTs based on different metrics using packets carrying information about the known state of the network: **Link-State Packets** (LSPs). These packets are forwarded when a node acquires new information.

In the LS algorithm, every node discovers the topology of the whole network and SPTs are calculated locally and independently by every node.

Complex event processing

Complex event processing systems add the ability to deploy rules that describes how composite events can be generated from primitive (or composite) ones.

Recently, a number of languages and systems have been proposed to support such architecture.



There are still some open issues with this architecture:

- **The rule language**: the rule language needs to find a balance between expressiveness and processing complexity
- **The processing engine**: how can the CEP engine efficiently match incoming events to build complex ones?
- **Distribution**: how can the processing be distributed in a network?

Stream-oriented communication

Stream-oriented communication is based on the notion of **data stream**: a sequence of data units organised in a standard way.

In stream-oriented communication, time does not usually impact the **correctness** of the communication, just its performance. This is not always the case however: e.g. when watching a live video feed.

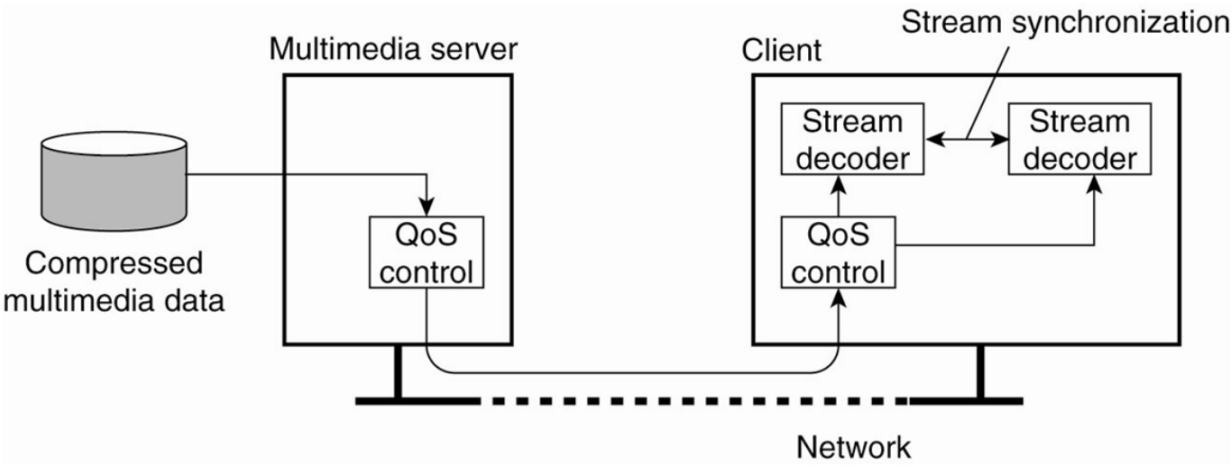
Stream-oriented communication has three main transmission modes:

- **Asynchronous**: the data items in a stream are transmitted one after the other without any further timing constraints (*apart ordering*)
- **Synchronous**: there is a maximum end-to-end delay for each unit in the data stream
- **Isochronous**: there is a maximum and a minimum end-to-end delay (*bounded jitter*)

Streams can be either **simple** or **complex**, the latter being composed of multiple sub-streams.

Non-functional requirements of steam-oriented communication are often expressed as **Quality of Service** requirements, like:

- Required bit rate
- Maximum delay to setup a session
- Maximum end-to-end delay
- Maximum variance in delay (a.k.a. **jitter**)



Quality of Service and the Internet: the DiffServ architecture

IP is a **best effort** protocol, but it offers a **Differentiated Services** field (a.k.a. **Type of Service** or **TOS**) in its header, which is composed of:

- 6 bits for the **Differentiated Services Code Point** (DSCP) field
- 2 bits for the **Explicit Congestion Notification** (ECP) field

The DSCP field encodes the **Per-Hop Behaviour** (PHB), which can be:

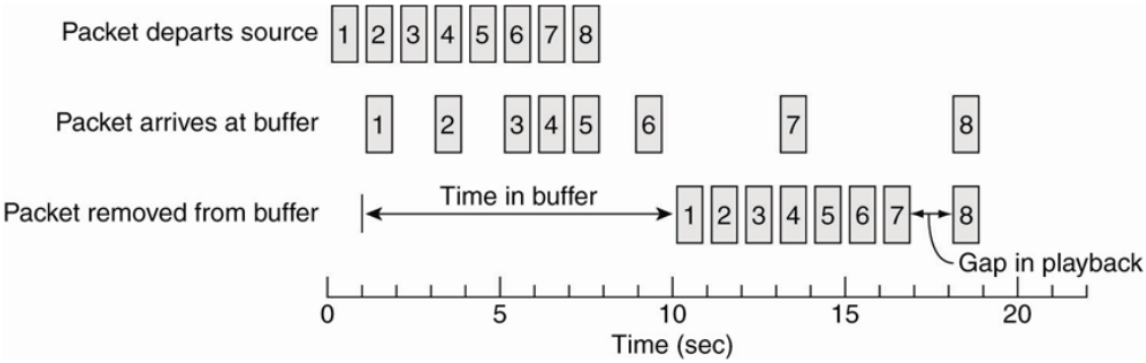
- Default
- Expedited forwarding
- Assured forwarding

The DiffServ field is not necessarily supported by Internet routers.

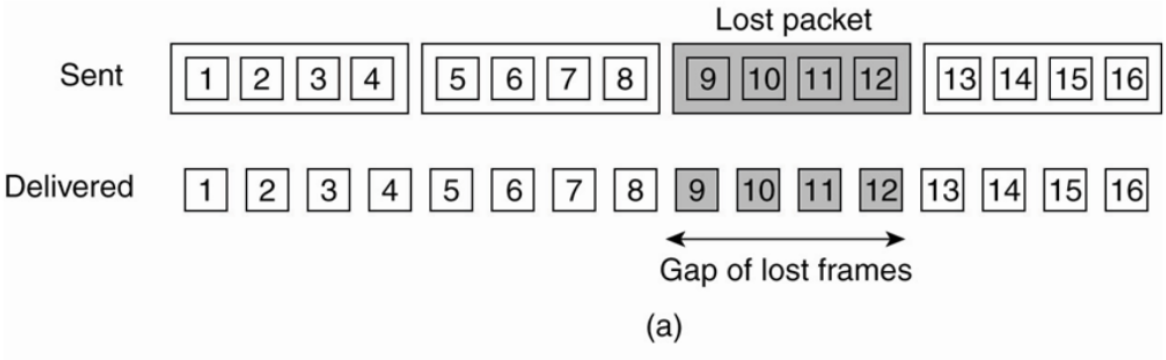
Enforcing QoS

Quality of Service can be enforced at the application layer with different techniques:

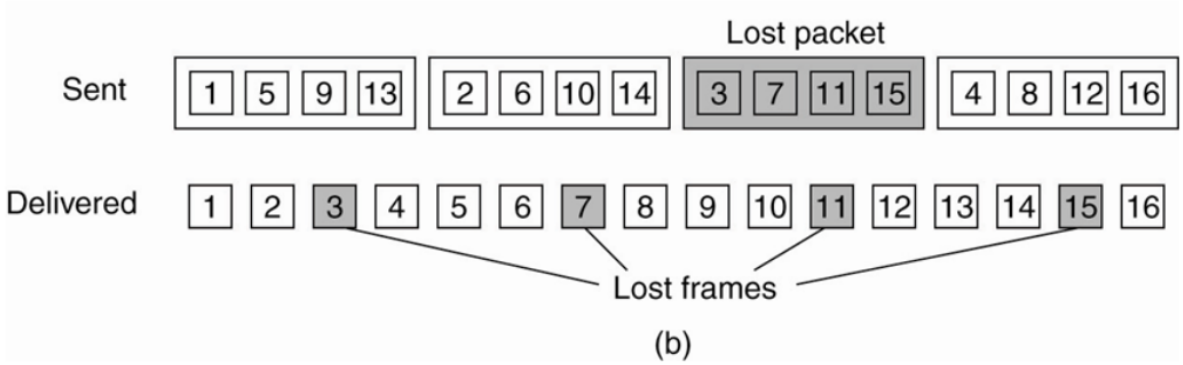
- **Buffering**: it can control the maximum jitter by sacrificing the session startup time



- **Forward Error Correction**



- **Interleaving data**: it can mitigate the impact of lost packets



Stream synchronisation

Synchronising two or more streams is not easy and could mean different things:

- Left and right channels at CD quality in an audio track means that each sample must be synchronised. This requires $23\mu s$ of maximum jitter
- Video and audio for lip synchronisation means that each audio interval must be in synch with its frame which, in turn, means that there can be at most $33ms$ of jitter

Synchronisation may take place at the sender or the receiver side. If the synchronisation happens at the sender side, the different streams can be merged to facilitate the operation.

Furthermore, stream synchronisation can happen at the application layer or at the middleware layer.